Automatic Machine Learning?

Andreas Müller

Columbia University, scikit-learn

Hey everybody and welcome to my talk about automatic machine learning. My name is Andreas Mueller, and I'm a lecturer at the columbia data science institute and I'm a core developer of the machine learning library scikit-learn. I want to talk to you about what automatic machine learning is, why we want it, and what the recent developments are. None of what I'll talk about is my work, but I hope I can give you a good overview of what's been happening in the area.

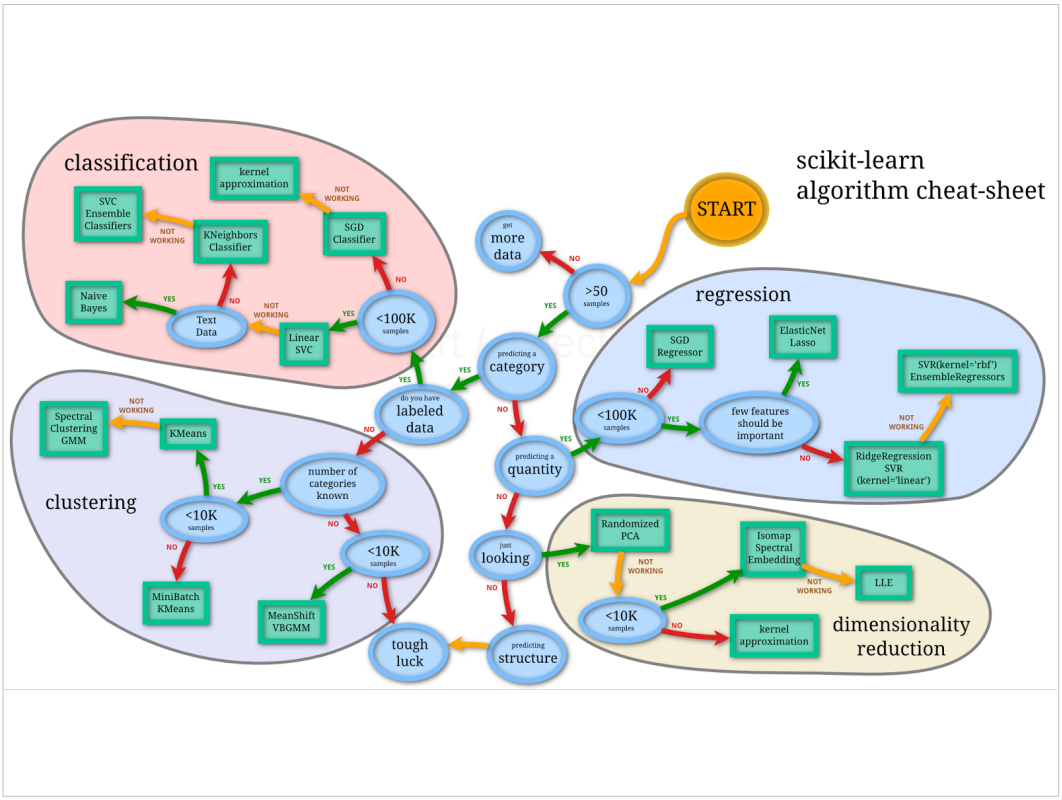
So first, why do I want automatic machine learning?

My personal goal is to make machine learning accessible and easy to use for everybody. Ideally, you shouldn’t need to read a book or understand a library or have a strong math background to apply machine learning to a problem. Unfortunately, that’s not where we are right now, and to apply machine learning successfully, you need significant knowledge in the field.

Issues with current tools (scikit-learn)

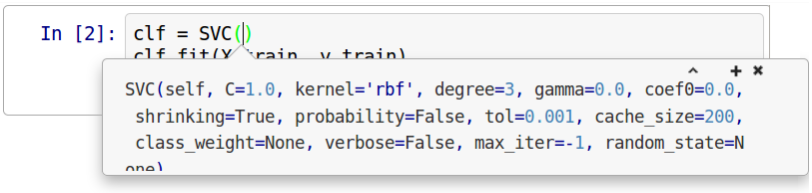
To start with, let me talk about where we are right now, in particular with respect to scikit-learn, which is arguably is the default machine learning library in python. What are the issues that beginners face, and how can we improve the situation?

I said none of this is my work, by which I meant the solutions. The problems might very well be somewhat my work.



Not sure if anyone remembers, but at some point, I made this ridiculous flow-chart. The idea is that it walks you through which machine learning model to pick for which task. It was a bit of a joke, because applying machine learning is much more complex than what the chart suggests. Still, people found it quite useful because it gave them at least **some** guidance on what kind of model to pick. Scikit-learn has an extensive user-guide, but it doesn't really give you a lot of advice on where to start. As a beginner, it's therefore hard to know which models to try first, and which models might be appropriate for a given task.

Selecting Hyper-Parameters



Even more tricky than selecting the model might be selecting the hyperparameters of the model. Most machine learning models have some parameters to tune to get the best performance. Scikit-learn models usually have many options, not all of which actually change the outcome. Many of them are trade-offs between runtime and accuracy, or very specific use-cases, and it's hard for a new-comer to know which of these to tune. Even if you know which parameters are important, you might still not know what a good range of values is, how many different values you should try, and whether you should try them on a linear or logarithmic scale.

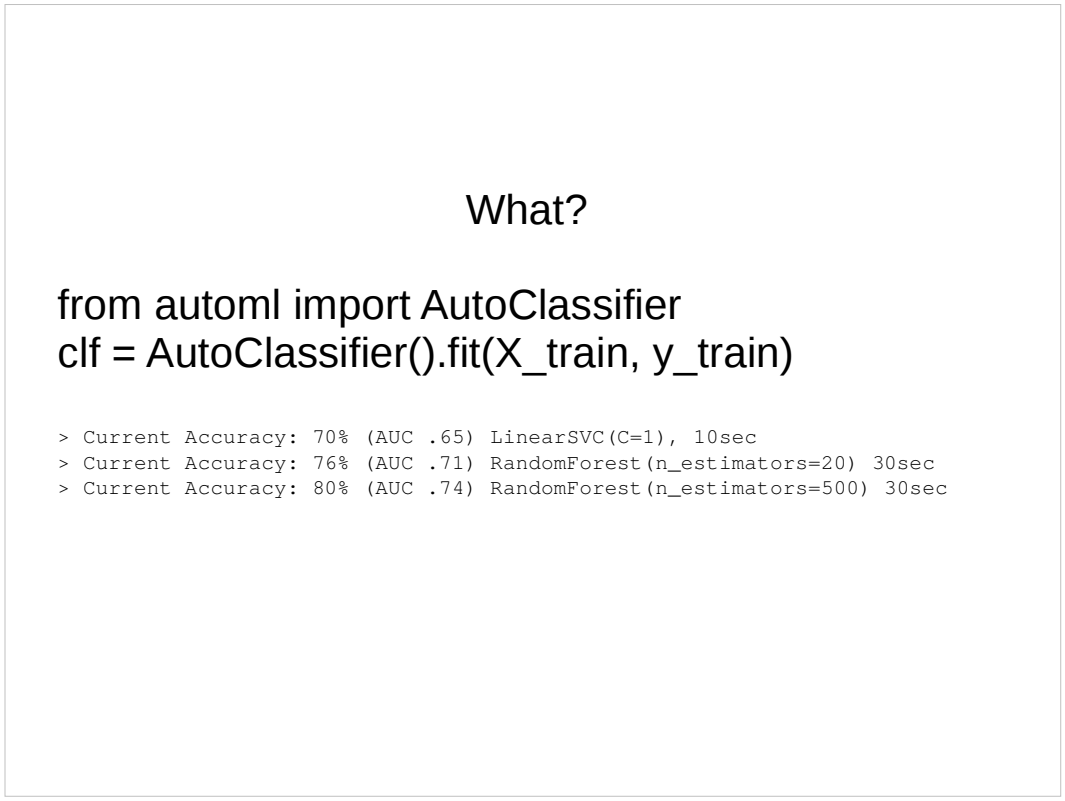
The same is true if you are not using scikit-learn but a deep learning library. Selecting and configuring a neural network can be quite challenging.

Scikit-learn: Explicit is better than implicit

```
make_pipeline(
    OneHotEncoder(),
    Imputer(),
    StandardScaler(),
    SVC())
```

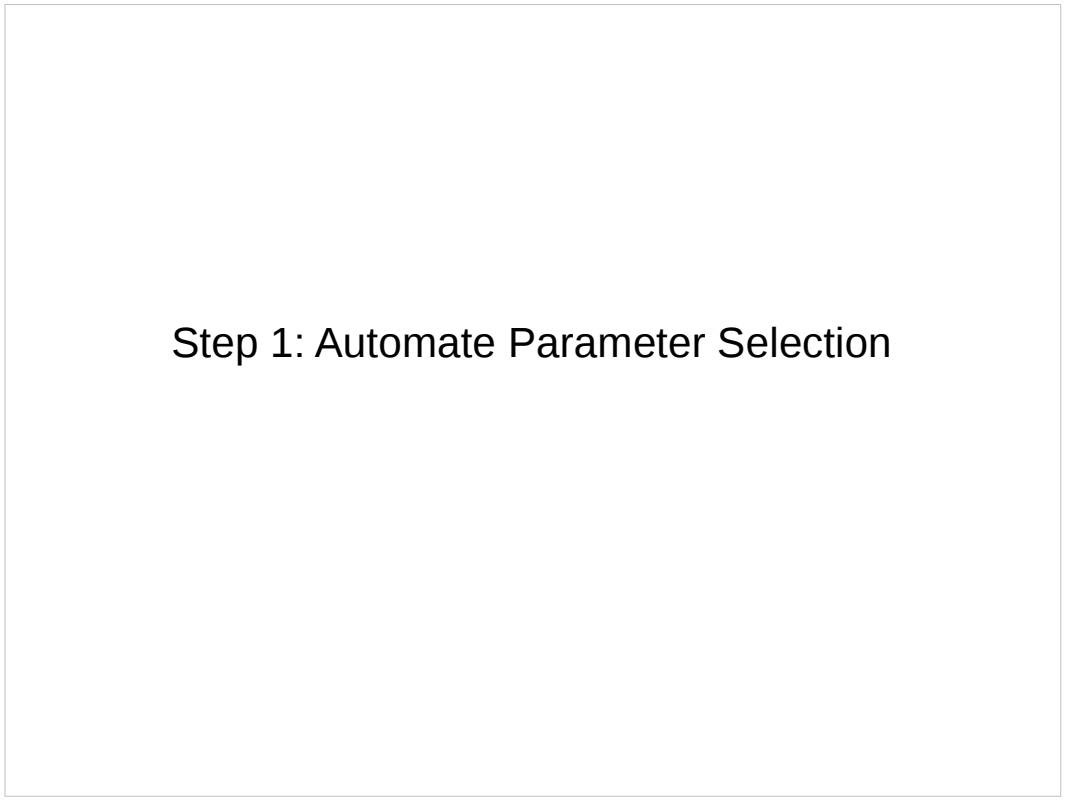
A final issue with the current situation, at least in Python, is that scikit-learn really likes to be very explicit. There is very little magic in the background, and you need to manually specify each data processing step. That’s a bit different in some R libraries, but that might give the user less control.

That means however, that you have to take care of many things yourself. For example dealing with categorical variables, with missing values and with scaling the data. So to apply a support vector machine to a dataset, you probably shouldn’t just fit the SVM model. Depending on your data, you might want to use a OneHotEncoder, Imputer and StandardScaler, each of which has some additional parameters. Beginners often don’t know about these best practices and might get bad results as a consequence.



Alright, so now that we have talked about the problem, what would we like our solution to look like? Here is my ideal interface. Import the automatic classifier, and when you fit it, it will provide you with a sequence of more and more accurate models over time, and you can stop at any point.

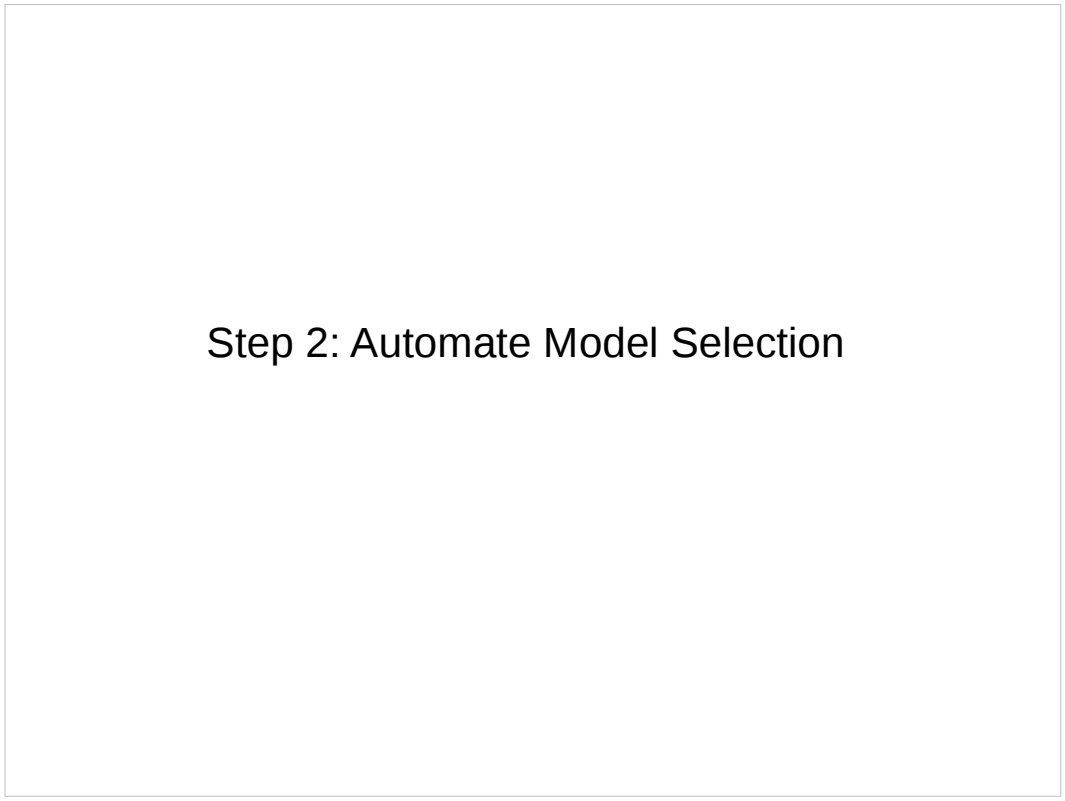
Actually, there is a library called auto-sklearn which contains an AutoClassifier that does something quite similar. It’s not entirely where I’d like it to be, though. I’ll talk about it more in a little bit.



So now that we know where we want to go, what are the steps we need to take? Step 1 for me is to automate the parameter selection. Given a particular model, I want my library to know which parameters are important, how they influence runtime and accuracy, and then search for the best parameter setting on my dataset in some smart way.

For example for a support vector machine, that could be the kernel, the parameters for the kernel and the regularization.

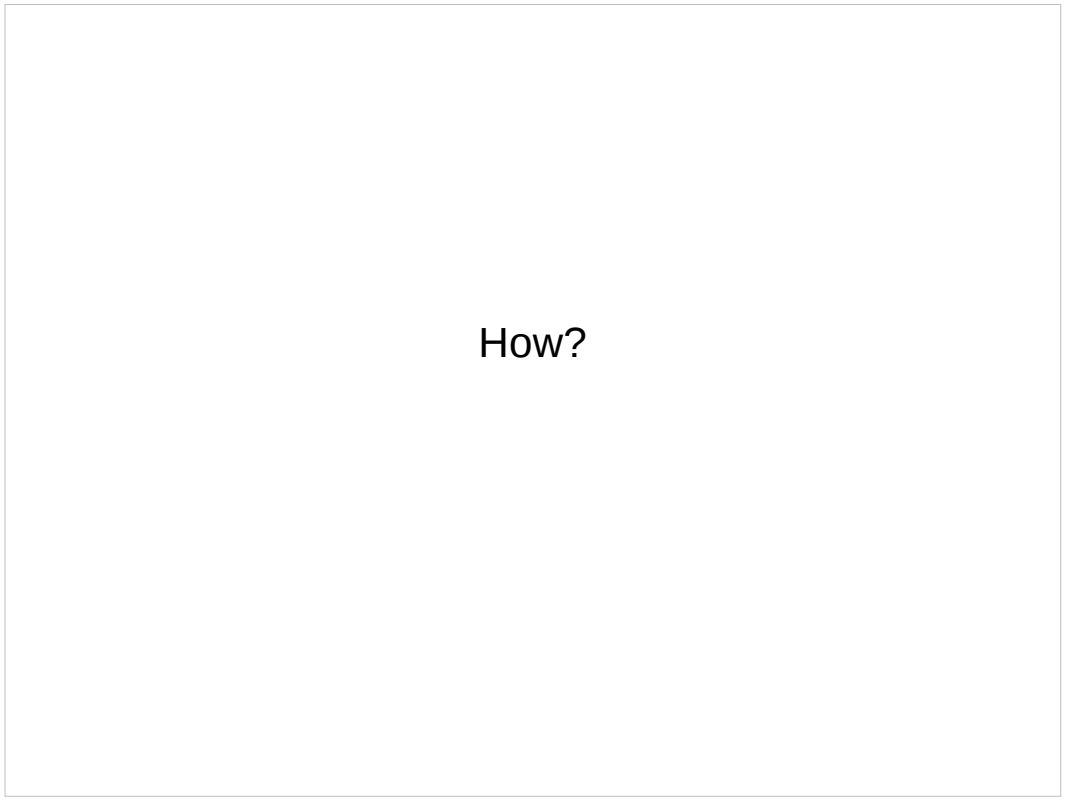
For a neural network that could be the number of layers, the non-linearities, the number of units per layer and the optimization algorithm.



The second step is to not only search the parameters, but also have a candidate list of models, and pick out the best model, together with the best parameters. This is sometimes called the “CASH” problem of Combined algorithm selection and hyperparameter optimization. You can go even one step further and ask that the library knows which models might be good for a particular dataset..

Step 3: Automate Pipeline Selection

Finally, we want not only the model to be selected automatically, but also all the preprocessing steps that the model might need.



So how do can we implement these three steps? How do we automate the parameter, model and pipeline selection?

Formalizing the Search Space

Discrete and Continuous Parameters
Conditional Parameters
Fixed pipeline vs flexible pipeline

First of all we need to formalize our search space. Even if we just want to select the parameters of one particular model, this search-space might be quite complex. There is often a mixture of discrete parameters, such as the kernel for a support vector machine, and continuous parameters, such as the amount of regularization. There is different ways to deal with continuous vs discrete parameters, which depend a lot on the search method we want to use. Another difficulty that arises already when just adjusting parameters for a single model is the existence of conditional parameters. For example for a neural network, the number of units in the second hidden layer only matters if we actually have at least two hidden layers.

Formalizing the Search Space

Discrete and Continuous Parameters
Conditional Parameters
Fixed pipeline vs flexible pipeline

If we want to search over whole pipelines, there is another difficulty, which is searching over the structure of the pipeline. Do we want fixed pipelines, or pipelines of fixed length? Should they be linear, or can we have more complex processing graphs?

All of these decisions are important, and again depend on our search strategy.

Finally, there is also the question on what to include in our search space. Should we include all models that are available and all parameters? Or only the most important ones?

Search Methods

Now, let’s talk about the different methods we can use to search the parameter spaces that we have defined.

Exhaustive Search (Grid Search)

The most standard one is exhaustive search, also known as grid-search in this context.

It’s easy to implement and understand and it’s embarrassingly parallel. Grid-Search can deal with conditional parameters if you specify them correctly.

However, you need to discretize all continuous parameters and provide explicit lists of parameters to try.

You can’t naturally trade-off accuracy and computational complexity in grid search, and when working with many different models or complex pipelines, exhaustive search quickly becomes infeasible.

Randomized Search

- Another straight-forward method is randomized search, which randomly samples the parameter for each model from a user-specified probability distribution. Randomized search can deal with conditional parameters, and is embarrassingly parallel, just as grid-search.
- Randomized search can naturally deal with continuous variables either by using continuous distributions, or by discretizing them as you would for grid-search. Depending on which of the two methods you choose, you either need to pick an appropriate continuous distribution, or an appropriate discretization.
- In contrast to Grid search, we can stop randomized search at any time, or just wait longer to get better results (This is known as an any-time algorithm).

Randomized Search

- A cool thing about randomized search is that for continuous distributions, the results will likely be better than grid search given the same amount of computation. That is because we are not using any regular structure to find the evaluation points.
- If some parameters are less important, that gets rid of a lot of redundant work.
- The paper by Bergstra and Bengio that I mentioned below gives more details.

Bayesian Optimization (SMBO)

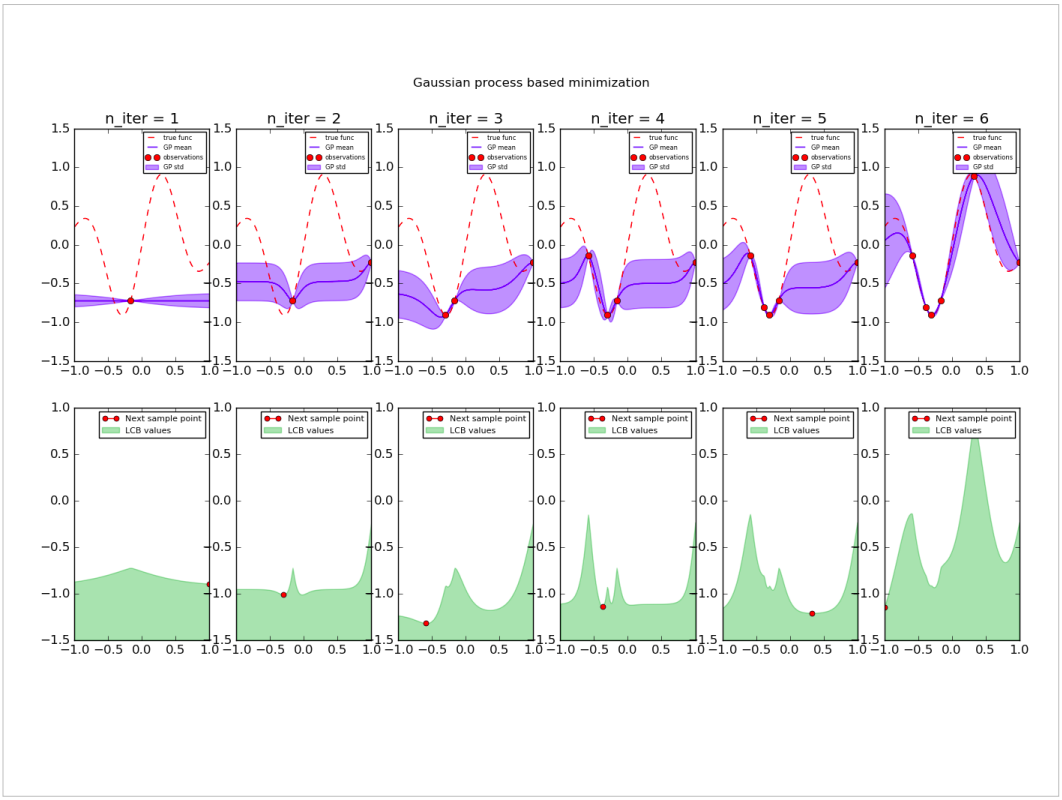
There is another, much more sophisticated family of methods for searching for parameters, known as sequential model based optimization or bayesian optimization. These are basically methods for global optimization for any expensive to compute, arbitrary function – such as the validation accuracy of a machine learning model.

The difference to other optimization methods that you might be familiar with like gradient descent or l-BFGS is that these optimization algorithms don't rely on gradients – which are usually not available for hyper-parameters, and that they are optimized for very slow functions – such as learning a neural network.

If evaluating your function takes days, you should spend much more effort in finding the next test point then simpler algorithms would do.

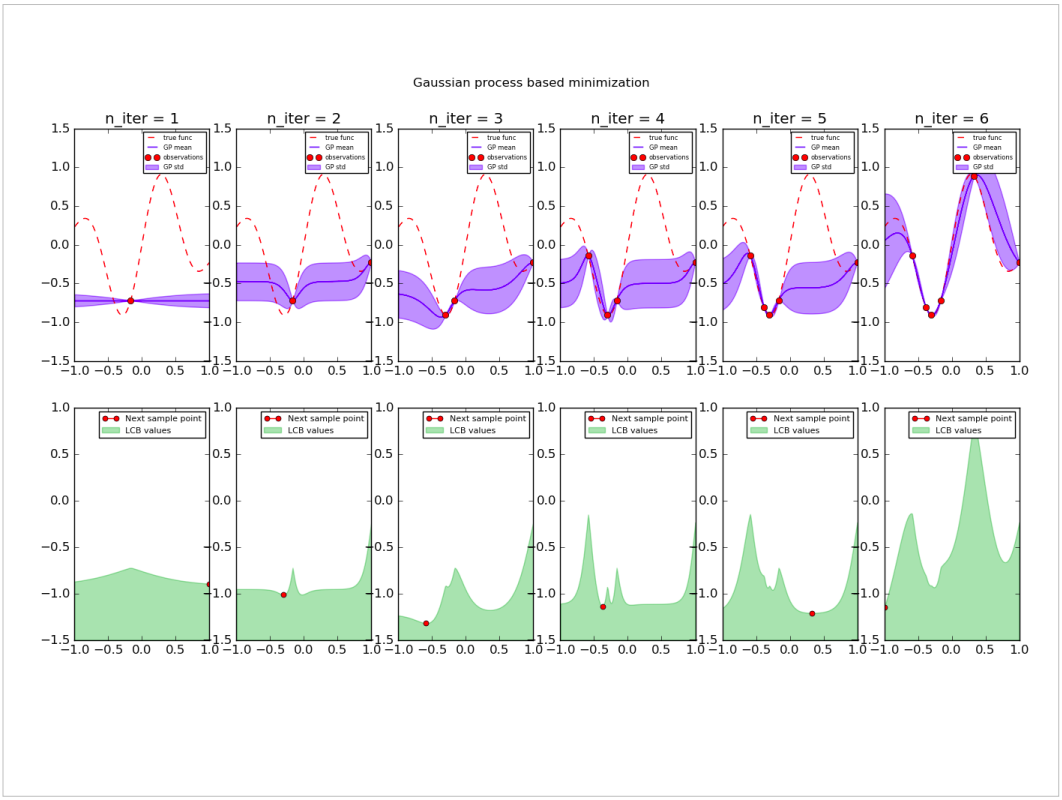
Bayesian Optimization (SMBO)

The idea behind all of the SMBO models is to build a probabilistic model of the mapping from parameter values to the validation accuracy for a particular model with a particular set of parameters. We iteratively select the most promising parameter values according to this model, build a model with these parameters and evaluate it. We can then use the validation performance to update our model of how the parameters influence accuracy.



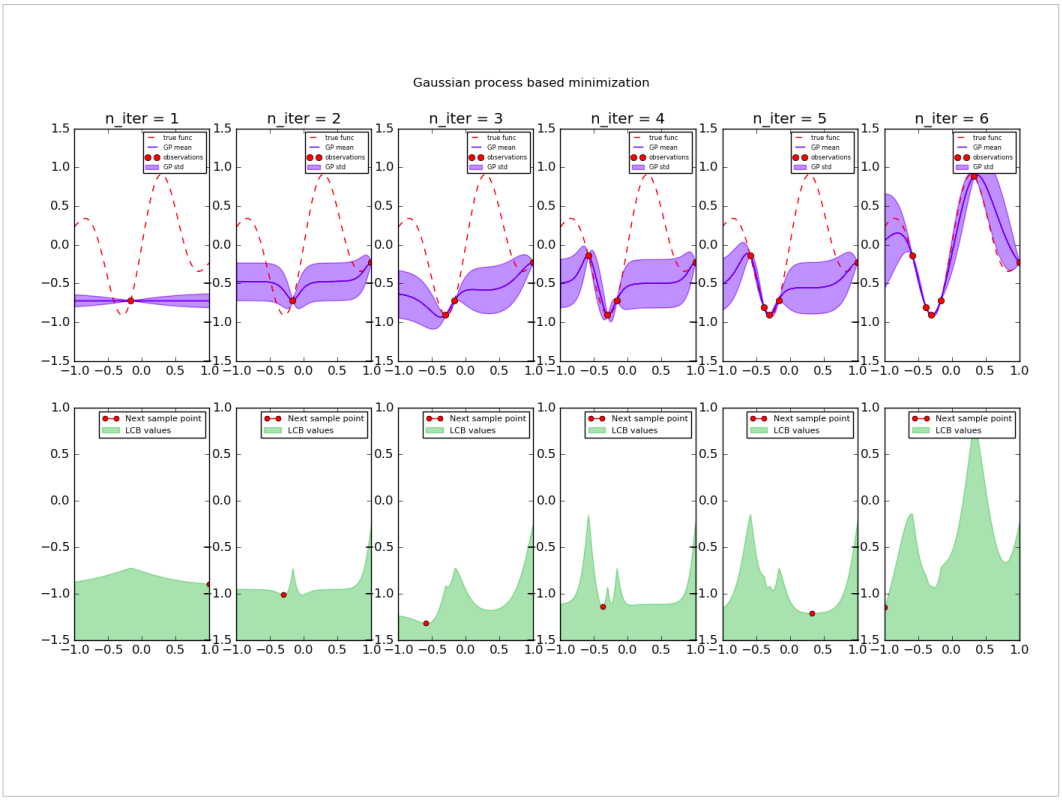
Here is an example of what this might look like when using a gaussian process for the probabilistic modelling. This is a toy example with a single parameter. The x-axis in these plots is the parameter setting and the y-axis is the error achieved with this setting.

In the top row, you can see as a red line the true dependency of the error on the parameter. We want to find the minimum of this function without access to its gradients. The red dots are observations, so parameter values that we tried and for which we know the error we can achieve. We picked the first one randomly. Then we use this point to build a surrogate model of the red function based on our observations. Here a gaussian process model is used, which you can see in purple, with mean and standard deviation.



Now we can use this surrogate to pick a new point to evaluate. We want to find a point that will give a low value, but we also want a point that will provide us with as much new information as possible. This is an instance of the exploration-exploitation trade-off.

One possible strategy is called lower confidence bound to find the most promising next point to try. In this strategy, we select the lowest point according to one standard deviation around the mean. This function is plotted in the bottom row. The minimum is on the very right corner, so we try this parameter setting, i.e. we do cross-validation of our model with this parameter setting. This yields another red point, as seen on the second panel. Then, we can update our surrogate model in purple to reflect the information we got from this point. Again, we find the most promising next point using the lower confidence bound, and so on. You can see that over time, we get closer and closer to the optimum, and get a better



The key point here is that in practice, evaluating the red function is very expensive, because it involves training a machine learning model, possibly on a very large dataset, so we want to try out as few parameter settings as possible. Evaluating the purple function, our model, or the green function, the so-called acquisition function, on the other hand is cheap. We also have gradients available for these functions – at least if we use Gaussian processes. So we can run a minimization algorithm on the cheap green function, and use the result to find promising parameter settings.

So now that we know the principles behind sequential model based optimization, let's look at the different variants that exist.

Common Models

	Discrete Parameters	Scalable	Conditional Parameters	Papers & implementations	Specify parameters
Gaussian Process	?	✗	?	many	bounds
Random Forest (SMAC)	✔	✔	✔	few	prior
Non-parametric (TPE)	✔	?	✔	few	prior

Using Gaussian processes, as we just did, is one of the most common approaches in the literature. It is well established and works quite well with continuous parameters. However, Gaussian processes can not really handle discrete parameters, and the commonly used models can't deal with conditional parameters. There is also issues with scaling to many evaluation points, as Gaussian processes can become expensive to evaluate when they are built using a large number of points.

Another approach uses random forests for building the model of the parameter influence. Random forests can work with continuous and discrete parameters, and can also accommodate conditional parameters. They scale well and seem to work relatively well in practice. The literature on the method is a bit limited, though, and there is only a single implementation out there. The method is very promising, though

Common Models

	Discrete Parameters	Scalable	Conditional Parameters	Papers & implementations	Specify parameters
Gaussian Process	?	✗	?	many	bounds
Random Forest (SMAC)	✔	✔	✔	few	prior
Non-parametric (TPE)	✔	?	✔	few	prior

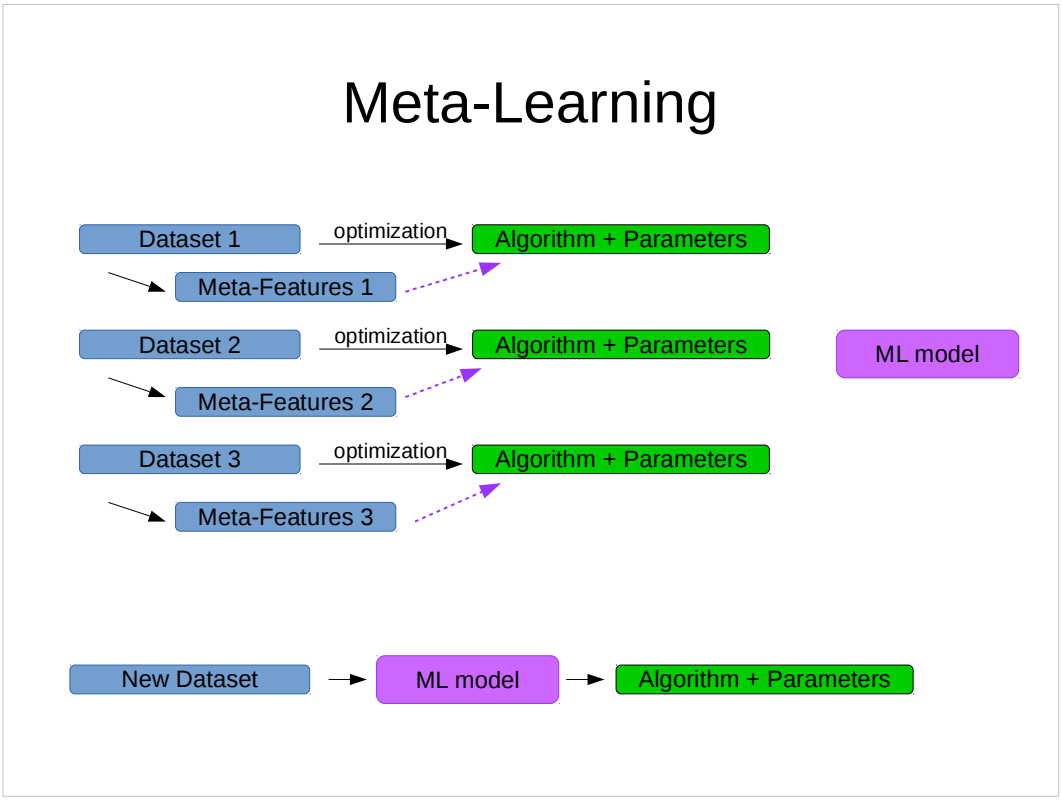
The Tree Parzen Window Estimate is a method developed by James Bergstra to specifically deal with conditional parameters. It seems to work reasonably well, but is not very widely adopted. Both, the random forest based approach and TPE require that you specify priors over parameters, while some advanced gaussian process methods only require bounds, which is somewhat more convenient.

I'll show you a quantitative comparison later.

Warm-starting and Meta-learning

Alright, so we talked about various strategies for searching for parameters or pipelines. However, a human data scientist wouldn’t start the search from scratch each time they encounter a new problem. So how can we jump-start this process?

This leads us to what is called “meta-learning” where you try to learn from a collection of datasets, and find out what algorithms work well on which kind of data.



So the idea of meta-learning is the following:

We start with some dataset and run our pipeline optimization algorithm, which finds a good algorithm and parameter setting. Then we do this again for a bunch of more datasets. This search usually takes quite a while, because the search space is quite big.

But once we have good parameters settings, we can use these settings to train a machine learning model. This is the “meta” model that tells us which algorithms are promising on a new dataset. To learn this model, we need to represent each dataset using what is called meta-features.

Then, to find a model for a new dataset, we just need to compute the meta-features, apply our meta-model, and will obtain a good model and parameters, much faster than doing the search all over again.

Meta-Features

The simplest of these meta-features are number of samples, number of features, and how sparse the data is. Then there is more advanced features like the moments of the dataset, or the mutual information between features and labels. Finally, so-called landmark features are very helpful. These features are of the type “how well does algorithm X” do on this dataset for some very fast to evaluate algorithm. Usually these are naive bayes and simple tree models, which are very fast to build.

Existing Approaches

So after talking about how to approach the problem of automatic machine learning in general, I want to give a brief overview of existing systems that try to solve this task.

auto-sklearn

(Hutter, Feurer, Eggensperger)

http://automl.github.io/auto-sklearn/stable/

Relatively unsurprisingly, my favorite of the existing projects is auto-sklearn, which attempts to automate scikit-learn pipelines. It is developed by a group in Freiburg, and you can get it on github and try it out. It uses a fixed setup for the pipline, with several optional steps like missing value imputation, scaling, feature selection and classification. It uses meta-learning using a nearest neighbor classifier. The outcome of the meta-learning is then used as a starting point for the model search, which is done using SMAC, the random forest based approach.

The package includes explicit distributions over parameters to search over for each model, which were hand-tuned.

Autoweka

http://www.cs.ubc.ca/labs/beta/Projects/autoweka/

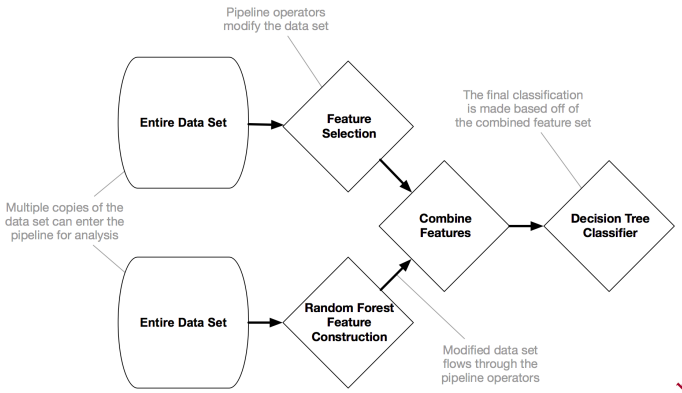
There is also autoweka, which is a plugin for weka that does model and parameter search. It allows for the use of both SMAC and TPE. Autoweka does not do meta-learning, and also does not construct more complex machine learning pipelines.

Hyperopt-sklearn

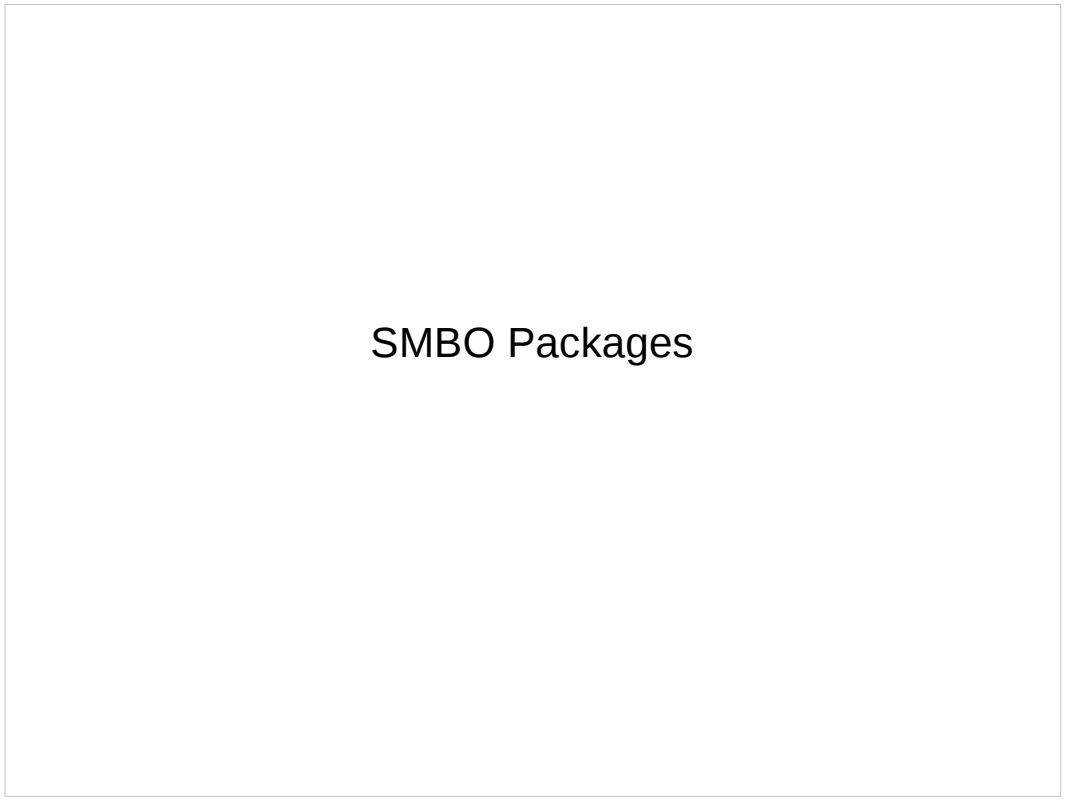
http://hyperopt.github.io/hyperopt-sklearn/

Hyperopt-sklearn is a project by the creator of TPE, James Bergstra, which basically provides a list of parameter settings for many of the scikit-learn models. These can then be optimized using hyperopt, his TPE implementation. Unfortunately this project is no longer maintained and has found very little adoption.

TPot



I also want to mention the Tpot project by my friend Randy Olson, who you probably follow on twitter. It uses genetic programming to evolve complex pipelines of scikit-learn models, together with their parameters. It contains dictionaries of parameter values to try, and is very very flexible in combining models and various mathematical functions for feature extraction. I haven't seen a benchmark against other methods that solve the CASH problem, yet, though.



These three are the only major automl open source packages that I’m aware of. There’s also a couple of commercial products out there, but given that they are very opaque, I’m not very interested in them.

I also want to list some packages that only solve the optimization problem, aka SMBO, as that is a very important part of the overall solution.

Spearmint

https://github.com/HIPS/Spearmint

Spearmint is a particular implementation of SMBO using gaussian processes, which works quite well. Unfortunately the authors created a startup, wetlab, and stopped developing the open version. I don’t think this library is very well maintained, though. If you want gaussian process based hyper parameter optimization, you probably want to look into the next one.

GPyOpt

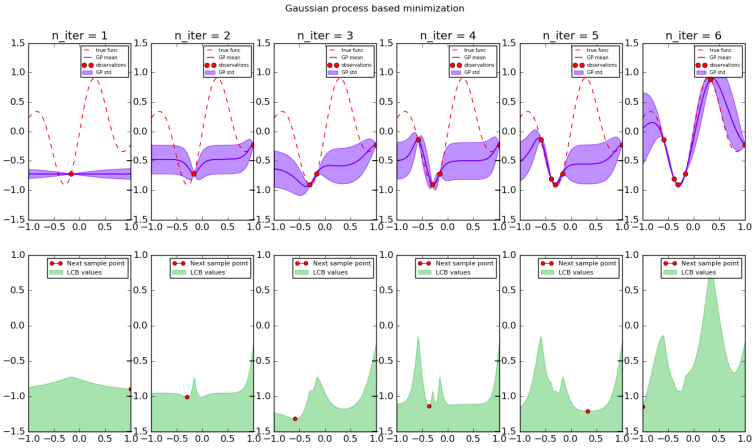
https://github.com/SheffieldML/GPyOpt

GpyOpt is an SMBO package using Gpy, a gaussian process library for Python. It comes out of the university of Sheffield, from Neil Lawrence’s group.

These guys are really the experts when it comes go Gaussian processes, and maybe you noticed that in the earlier benchmark, their system won.

They are also pretty good when it comes to software engineering aspects.

Scikit-optimize



Some of the scikit-learn developers have started working on scikit-optimize, not to be confused with scikit-optimization.

Scikit-optimize currently implements a GP based, random forest based and gradient boosting based methods. They try to create robust reference implementations for the different approaches and the project is fairly active right now. It also contains a drop-in replacement for GridSearchCV in scikit-learn.

Some Benchmarks

Experiment	# Evals	SMAC	TPE	Spearmint	DNGO
Branin (0.398)	200	0.655 ± 0.27	0.526 ± 0.13	$\mathbf{0.398 \pm 0.00}$	$\mathbf{0.398 \pm 0.00}$
Hartmann6 (-3.322)	200	-2.977 ± 0.11	-2.823 ± 0.18	$\mathbf{-3.3166 \pm 0.02}$	-3.319 ± 0.00
Logistic Regression	100	8.6 ± 0.9	8.2 ± 0.6	$\mathbf{6.88 \pm 0.0}$	$\mathbf{6.89 \pm 0.04}$
LDA (On grid)	50	1269.6 ± 2.9	1271.5 ± 3.5	$\mathbf{1266.2 \pm 0.1}$	$\mathbf{1266.2 \pm 0.0}$
SVM (On grid)	100	$\mathbf{24.1 \pm 0.1}$	24.2 ± 0.0	$\mathbf{24.1 \pm 0.1}$	$\mathbf{24.1 \pm 0.1}$

Here is a quantitative comparison of these search methods on some simple benchmark datasets. In this comparison, the number of evaluations of the expensive function is fixed, and we look at how good of a minimum each of the approaches finds. Lower is better. Spearmint here is a particular implementation of a Gaussian process based approach. The method on the very right is one that I didn't talk about, but it's done by the same people that created spearmint. So it's not entirely surprising that spearmint,is the best in all of these. However, remember it doesn't really work with conditional parameters, so it might not be that great for us. SMAC and TPE do both reasonably well, with no clear winner.

Results below are for `n_calls=64`:

Method	Average rank (less is better)
dummy_minimize	4.552
forest_minimize	2.362
gbrt_minimize	2.172
gp_minimize	1.241
gpyopt_minimize	1.069
hyperopt_minimize	3.052
smac_minimize	3.431

scikit-optimize

GpyOpt (GP)

Hyperopt (TPE)

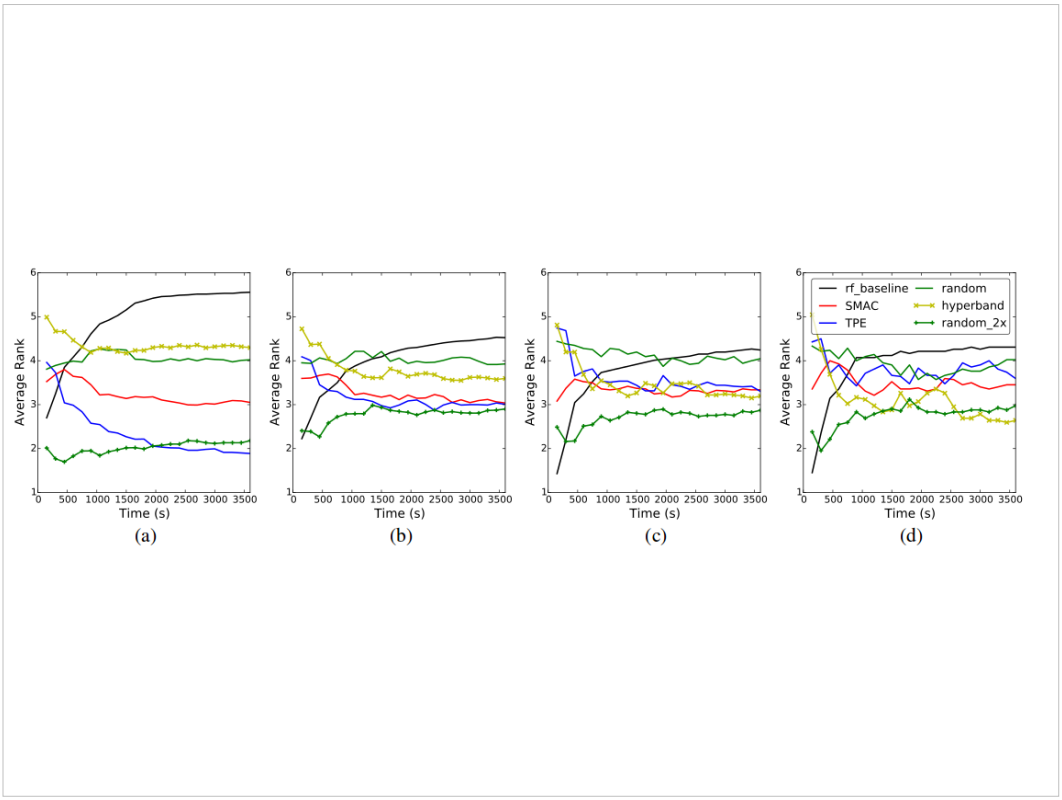
SMAC (Random Forest)

https://github.com/iaroslav-ai/scikit-optimize-benchmarks

Here is a similar benchmark, this time by the authors of scikit-optimize. This aggregates over multiple benchmark problems. The easiest way to aggregate is to look at the average relative rank of the algorithms after a fixed number of iterations, where lower rank is better.

Dummy minimize at the top is just random sampling.

You can see that in this benchmark, scikit-optimize does fairly well, but GpyOpt wins.



Here is another interesting comparison, which includes “hyperband”, which you shouldn’t worry about for now. It shows the rank of each optimization procedure over time. Lower is better, you want your algorithm to be the best, so ranked first at all times. Here the authors also compared against randomized search in green, which is basically entirely “stupid” because it has no model.

Random search mostly does worse than the rest, but they also compared against random search that can do two evaluations for each evaluation that the other algorithms get, so basically giving random search twice as much time. That’s the green line at the bottom. It is clearly better than all the others. What that means is basically that the complicated methods are less than twice as fast as using random search, which is interesting.

Within Scikit-learn

- GridSearchCV
- RandomizedSearchCV
- Searching over Pipelines
- Built-in parameter ranges (coming)

Finally, there are parts of the problem that are or will be addressed in scikit-learn. We already have the brute force search and the randomized search.

We also have the capability to search over the steps in a pipeline.

What I want to get to soon is to add some built-in parameter ranges, so that the users, or our algorithms, know what to search over.

TODO

Clean separation of:
- Model Search Space
- Pipeline Search Space
- Optimization Method
- Meta-Learning

- Exploit prior knowledge better!
- Usability
- Runtime consideration

So now you might think that all the work has already been done, but I think we are just at the beginning. Here is what I think we should do next. First, we should have a clean separation of our definition of the model search space, the pipeline search space, the optimization method and the meta-learning. These components are only slightly coupled, and working on only one at a time makes comparing different approaches much simpler.

Additionally, I think the current meta-learning approaches don’t take full advantage of the information available in the training data, and we can do better. There is also a lot to do in terms of usability, as most of these packages are research software, and might not be easy to get to work.

TODO

Clean separation of:
- Model Search Space
- Pipeline Search Space
- Optimization Method
- Meta-Learning

- Exploit prior knowledge better!
- Usability
- Runtime consideration
- Data subsampling

Furthermore, I don’t know of any methods that take the runtime of algorithms into account when doing their search. Usually want to try the fast methods first. There is lost of papers on the topic, but I don’t know of any implementation in the context of auto-ml.

Finally, most of the current methods don’t use subsampling of the dataset for faster evaluation. This is only implemented in the hyperband algorithm which I mentioned briefly. I think subsampling datasets in the early stages of the search is essential for good performance in any real-world system.

Criticism

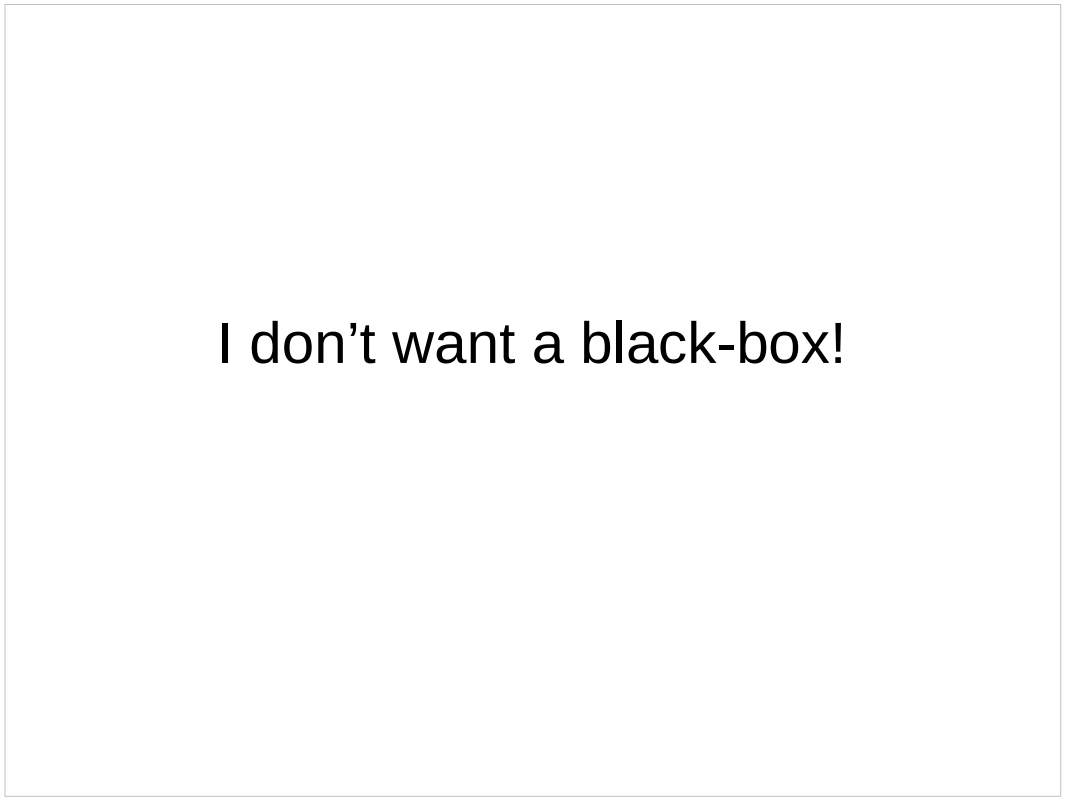
So now that I outlined what I think we should do, I want to mention and address some of the criticisms of auto-ml. I think auto-ml is the or at least a way forward, but there are some caveats, and I’m not sure if the current systems take the right approach.

Randomized Search works well

This is actually something I’ve been saying for years – and James Bergstra has the paper to prove it. Randomized search works quite well. From the findings of the recent paper that I showed some plots from, it looks like the best approaches are about 1.5 times as fast as randomized search. So maybe randomized search is a good candidate for an initial implementation, as it is so simple to implement. However, with more complex spaces, the advantage of the complex approaches might increase. Also, a 1.5 times speedup is not too bad, if the user can get it for free if we implement it for them.

Do we need 100 Classifiers?
Do we need Complex pipelines?

One central question for me is: Do we need that many models, do we need complex pipelines? This is actually the thing I’m most worried about. Some papers use a random forest as the baseline for their algorithms, and the random forest actually doesn’t do too badly. If you add in gradient boosting, and maybe logistic regression, perhaps we can beat the methods with the giant search spaces? I think we know way too little about which kind of methods work well on which kind of data, and we should really think about how large a search space we need, to do well on the majority of tasks.



Even if we succeed, automl might not be the cure-all solution. In many applications, people don’t want black box algorithms that are not interpretable. Auto-sklearn for example improves performance by ensembling the k top performing models. So this is an ensemble of automatically optimized pipelines. Understanding this final model is quite non-trivial. Maybe there is a way to penalize complex pipelines, or a way to restrict ourself to methods that provide interpretable results.

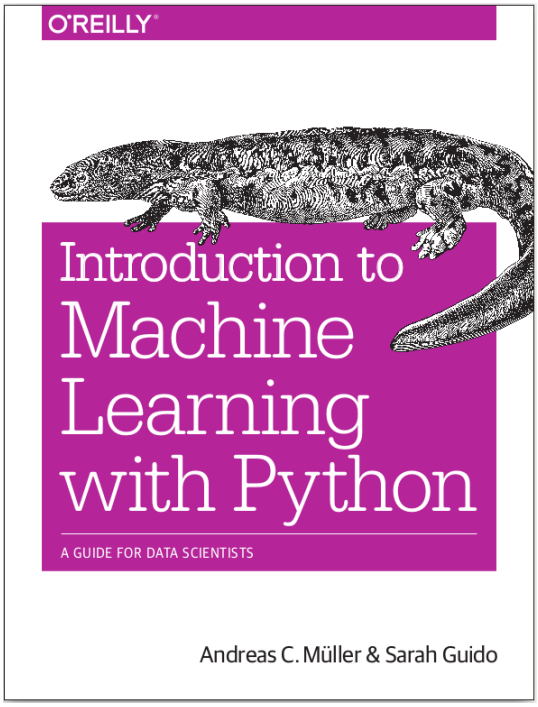
Still, there are many applications in which a black box is just fine.

Making it too easy?

- One somewhat common criticism is that this might make machine learning “too easy” and so people will abuse it, or use it without knowing what the results mean, or without taking proper care in evaluating the results. Some people already criticized even scikit-learn, that still needs a lot of manual work, for making it “too easy”. I don’t think there’s such a thing as “too easy to use”, but you probably need a minimum training in data science or statistics to make reasonable use of any automated learning algorithm if you want to apply it to a new domain.
- However, I think we should communicate good practices and provide resources for people to educate themselves.
- I also think that more automated algorithms can have more built-in safety mechanisms, that can enforce best practices. So maybe this will even be a step forward.

Material

- Taking the Human Out of the Loop: A Review of Bayesian Optimization (Shahriari, Swersky, Wang, Adams, de Freitas)
- Random Search for Hyper-Parameter Optimization (Bergstra, Bengio)
- Efficient and Robust Automated Machine Learning (Feurer et al) [autosklearn]
- http://automl.github.io/auto-sklearn/stable/
- Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits (Lie et. al) [hyperband] https://arxiv.org/abs/1603.06560
- Scalable Bayesian Optimization Using Deep Neural Networks [Snoek et al]
- https://github.com/iaroslav-ai/scikit-optimize-benchmarks



amueller.github.io

@amuellerml

@amueller

t3kcit@gmail.com

https://github.com/amueller/talks_odt/

Buy my book. It's about machine learning. It's written for programmers, and should be a pretty easy read for anyone that knows a bit of numpy. I avoided adding to much math. If you want math, there are many awesome books to check out.

You can find the slides and notes for this talk in the github repo I listed at the bottom.